- Sets the locale according to the environment. On ANSI C-based systems this is done by calling **setlocale**(**LC_ALL**, *language*). If an error is encountered, a warning message is issued with **XtWarning**.
- Calls **XSupportsLocale** to verify that the current locale is supported. If the locale is not supported, a warning message is issued with **XtWarning** and the locale is set to “C”.
- Calls **XSetLocaleModifiers** specifying the empty string.
- Returns the value of the current locale. On ANSI C-based systems this is the return value from a final call to **setlocale**(**LC_ALL**, NULL).

A client wishing to use this mechanism to establish locale can do so by calling **XtSetLanguage-Proc** prior to **XtDisplayInitialize**, as in the following example.

```
Widget top;
XtSetLanguageProc(NULL, NULL, NULL);
top = XtOpenApplication(...);
...
```

2.3. Loading the Resource Database

The **XtDisplayInitialize** function first determines the language string to be used for the specified display. It then creates a resource database for the default screen of the display by combining the following sources in order, with the entries in the first named source having highest precedence:

- Application command line (*argc*, *argv*).
- Per-host user environment resource file on the local host.
- Per-screen resource specifications from the server.
- Per-display resource specifications from the server or from the user preference file on the local host.
- Application-specific user resource file on the local host.
- Application-specific class resource file on the local host.

When the resource database for a particular screen on the display is needed (either internally, or when **XtScreenDatabase** is called), it is created in the following manner using the sources listed above in the same order:

- A temporary database, the “server resource database”, is created from the string returned by **XResourceManagerString** or, if **XResourceManagerString** returns NULL, the contents of a resource file in the user’s home directory. On POSIX-based systems, the usual name for this user preference resource file is $HOME/**.Xdefaults**.

- If a language procedure has been set, **XtDisplayInitialize** first searches the command line for the option “-xnlLanguage”, or for a -xrm option that specifies the xnlLanguage/XnlLanguage resource, as specified by Section 2.4. If such a resource is found, the value is assumed to be entirely in XPCS, the X Portable Character Set. If neither option is specified on the command line, **XtDisplayInitialize** queries the server resource database (which is assumed to be entirely in XPCS) for the resource *name***.xnlLanguage**, class *Class***.XnlLanguage** where *name* and *Class* are the *application_name* and *application_class* specified to **XtDisplayInitialize**. The language procedure is then invoked with the resource value if found, else the empty string. The string returned from the language procedure is saved for all future references in the Intrinsics that require the per-display language string.

- The screen resource database is initialized by parsing the command line in the manner specified by Section 2.4.

- If a language procedure has not been set, the initial database is then queried for the resource *name*.**xnlLanguage**, class *Class*.**XnlLanguage** as specified above. If this database query fails, the server resource database is queried; if this query also fails, the language is determined from the environment; on POSIX-based systems, this is done by retrieving the value of the **LANG** environment variable. If no language string is found, the empty string is used. This language string is saved for all future references in the Intrinsics that require the per-display language string.

- After determining the language string, the user's environment resource file is then merged into the initial resource database if the file exists. This file is user-, host-, and process-specific and is expected to contain user preferences that are to override those specifications in the per-display and per-screen resources. On POSIX-based systems, the user's environment resource file name is specified by the value of the **XENVIRONMENT** environment variable. If this environment variable does not exist, the user's home directory is searched for a file named **.Xdefaults-***host*, where *host* is the host name of the machine on which the application is running.

- The per-screen resource specifications are then merged into the screen resource database, if they exist. These specifications are the string returned by **XScreenResourceString** for the respective screen and are owned entirely by the user.

- Next, the server resource database created earlier is merged into the screen resource database. The server property, and corresponding user preference file, are owned and constructed entirely by the user.

- The application-specific user resource file from the local host is then merged into the screen resource database. This file contains user customizations and is stored in a directory owned by the user. Either the user or the application or both can store resource specifications in the file. Each should be prepared to find and respect entries made by the other. The file name is found by calling **XrmSetDatabase** with the current screen resource database, after preserving the original display-associated database, then calling **XtResolvePathname** with the parameters (*display*, NULL, NULL, NULL, *path*, NULL, 0, NULL), where *path* is defined in an operating-system-specific way. On POSIX-based systems, *path* is defined to be the value of the environment variable **XUSERFILESEARCHPATH** if this is defined. If **XUSERFILESEARCHPATH** is not defined, an implementation-dependent default value is used. This default value is constrained in the following manner:

  – If the environment variable **XAPPLRESDIR** is not defined, the default **XUSERFILESEARCHPATH** must contain at least six entries. These entries must contain $HOME as the directory prefix, plus the following substitutions:

1. %C, %N, %L	or	%C, %N, %l, %t, %c
2. %C, %N, %l		
3. %C, %N		
4. %N, %L	or	%N, %l, %t, %c
5. %N, %l		
6. %N		

  The order of these six entries within the path must be as given above. The order and use of substitutions within a given entry are implementation-dependent.

– If **XAPPLRESDIR** is defined, the default **XUSERFILESEARCHPATH** must contain at least seven entries. These entries must contain the following directory prefixes and substitutions:

1.	$XAPPLRESDIR	with	%C, %N, %L	or	%C, %N, %l, %t, %c
2.	$XAPPLRESDIR	with	%C, %N, %l		
3.	$XAPPLRESDIR	with	%C, %N		
4.	$XAPPLRESDIR	with	%N, %L	or	%N, %l, %t, %c
5.	$XAPPLRESDIR	with	%N, %l		
6.	$XAPPLRESDIR	with	%N		
7.	$HOME	with	%N		

The order of these seven entries within the path must be as given above. The order and use of substitutions within a given entry are implementation-dependent.

• Last, the application-specific class resource file from the local host is merged into the screen resource database. This file is owned by the application and is usually installed in a system directory when the application is installed. It may contain sitewide customizations specified by the system manager. The name of the application class resource file is found by calling **XtResolvePathname** with the parameters (*display*, ‘‘app-defaults’’, NULL, NULL, NULL, NULL, 0, NULL). This file is expected to be provided by the developer of the application and may be required for the application to function properly. A simple application that wants to be assured of having a minimal set of resources in the absence of its class resource file can declare fallback resource specifications with **XtAppSetFallbackResources**. Note that the customization substitution string is retrieved dynamically by **XtResolvePathname** so that the resolved file name of the application class resource file can be affected by any of the earlier sources for the screen resource database, even though the contents of the class resource file have lowest precedence. After calling **XtResolvePathname**, the original display-associated database is restored.

To obtain the resource database for a particular screen, use **XtScreenDatabase**.

XrmDatabase XtScreenDatabase(*screen*)
    Screen **screen*;

screen Specifies the screen whose resource database is to be returned.

The **XtScreenDatabase** function returns the fully merged resource database as specified above, associated with the specified screen. If the specified *screen* does not belong to a **Display** initialized by **XtDisplayInitialize**, the results are undefined.

To obtain the default resource database associated with a particular display, use **XtDatabase**.

XrmDatabase XtDatabase(*display*)
    Display **display*;

display Specifies the display.

The **XtDatabase** function is equivalent to **XrmGetDatabase**. It returns the database associated with the specified display, or NULL if a database has not been set.

To specify a default set of resource values that will be used to initialize the resource database if no application-specific class resource file is found (the last of the six sources listed above), use **XtAppSetFallbackResources**.

void XtAppSetFallbackResources(*app_context*, *specification_list*)
    XtAppContext *app_context*;
    String **specification_list*;

app_context	Specifies the application context in which the fallback specifications will be used.
specification_list	Specifies a NULL-terminated list of resource specifications to preload the database, or NULL.

Each entry in *specification_list* points to a string in the format of **XrmPutLineResource**. Following a call to **XtAppSetFallbackResources**, when a resource database is being created for a particular screen and the Intrinsics are not able to find or read an application-specific class resource file according to the rules given above and if *specification_list* is not NULL, the resource specifications in *specification_list* will be merged into the screen resource database in place of the application-specific class resource file. **XtAppSetFallbackResources** is not required to copy *specification_list*; the caller must ensure that the contents of the list and of the strings addressed by the list remain valid until all displays are initialized or until **XtAppSetFallbackResources** is called again. The value NULL for *specification_list* removes any previous fallback resource specification for the application context. The intended use for fallback resources is to provide a minimal number of resources that will make the application usable (or at least terminate with helpful diagnostic messages) when some problem exists in finding and loading the application defaults file.

2.4. Parsing the Command Line

The **XtOpenDisplay** function first parses the command line for the following options:

−display	Specifies the display name for **XOpenDisplay**.
−name	Sets the resource name prefix, which overrides the application name passed to **XtOpenDisplay**.
−xnllanguage	Specifies the initial language string for establishing locale and for finding application class resource files.

XtDisplayInitialize has a table of standard command line options that are passed to **XrmParseCommand** for adding resources to the resource database, and it takes as a parameter additional application-specific resource abbreviations. The format of this table is described in Section 15.9 in *Xlib — C Language X Interface*.

```
typedef enum {
        XrmoptionNoArg,          /* Value is specified in OptionDescRec.value */
        XrmoptionIsArg,          /* Value is the option string itself */
        XrmoptionStickyArg,      /* Value is characters immediately following option */
        XrmoptionSepArg,         /* Value is next argument in argv */
        XrmoptionResArg,         /* Use the next argument as input to XrmPutLineResource*/
        XrmoptionSkipArg,        /* Ignore this option and the next argument in argv */
        XrmoptionSkipNArgs,      /* Ignore this option and the next */
                                 /* OptionDescRec.value arguments in argv */
        XrmoptionSkipLine        /* Ignore this option and the rest of argv */
} XrmOptionKind;

typedef struct {
        char *option;            /* Option name in argv */
        char *specifier;         /* Resource name (without application name) */
        XrmOptionKind argKind;   /* Location of the resource value */
        XPointer value;          /* Value to provide if XrmoptionNoArg */
} XrmOptionDescRec, *XrmOptionDescList;
```

The standard table contains the following entries:

Option String	Resource Name	Argument Kind	Resource Value
−background	*background	SepArg	next argument
−bd	*borderColor	SepArg	next argument
−bg	*background	SepArg	next argument
−borderwidth	.borderWidth	SepArg	next argument
−bordercolor	*borderColor	SepArg	next argument
−bw	.borderWidth	SepArg	next argument
−display	.display	SepArg	next argument
−fg	*foreground	SepArg	next argument
−fn	*font	SepArg	next argument
−font	*font	SepArg	next argument
−foreground	*foreground	SepArg	next argument
−geometry	.geometry	SepArg	next argument
−iconic	.iconic	NoArg	“true”
−name	.name	SepArg	next argument
−reverse	.reverseVideo	NoArg	“on”
−rv	.reverseVideo	NoArg	“on”
+rv	.reverseVideo	NoArg	“off”
−selectionTimeout	.selectionTimeout	SepArg	next argument
−synchronous	.synchronous	NoArg	“on”
+synchronous	.synchronous	NoArg	“off”
−title	.title	SepArg	next argument
−xnllanguage	.xnlLanguage	SepArg	next argument
−xrm	next argument	ResArg	next argument
−xtsessionID	.sessionID	SepArg	next argument

Note that any unique abbreviation for an option name in the standard table or in the application table is accepted.

If reverseVideo is **True**, the values of **XtDefaultForeground** and **XtDefaultBackground** are exchanged for all screens on the Display.

The value of the synchronous resource specifies whether or not Xlib is put into synchronous mode. If a value is found in the resource database during display initialization, **XtDisplayInitialize** makes a call to **XSynchronize** for all display connections currently open in the application context. Therefore, when multiple displays are initialized in the same application context, the most recent value specified for the synchronous resource is used for all displays in the application context.

The value of the selectionTimeout resource applies to all displays opened in the same application context. When multiple displays are initialized in the same application context, the most recent value specified is used for all displays in the application context.

The −xrm option provides a method of setting any resource in an application. The next argument should be a quoted string identical in format to a line in the user resource file. For example, to give a red background to all command buttons in an application named **xmh**, you can start it up as

```
xmh −xrm ’xmh*Command.background: red’
```

When it parses the command line, **XtDisplayInitialize** merges the application option table with the standard option table before calling the Xlib **XrmParseCommand** function. An entry in the application table with the same name as an entry in the standard table overrides the standard table entry. If an option name is a prefix of another option name, both names are kept in the merged table. The Intrinsics reserve all option names beginning with the characters ‘‘-xt’’ for future standard uses.

2.5. Creating Widgets

The creation of widget instances is a three-phase process:

1. The widgets are allocated and initialized with resources and are optionally added to the managed subset of their parent.
2. All composite widgets are notified of their managed children in a bottom-up traversal of the widget tree.
3. The widgets create X windows, which then are mapped.

To start the first phase, the application calls **XtCreateWidget** for all its widgets and adds some (usually, most or all) of its widgets to their respective parents’ managed set by calling **XtManageChild**. To avoid an $O(n^2)$ creation process where each composite widget lays itself out each time a widget is created and managed, parent widgets are not notified of changes in their managed set during this phase.

After all widgets have been created, the application calls **XtRealizeWidget** with the top-level widget to execute the second and third phases. **XtRealizeWidget** first recursively traverses the widget tree in a postorder (bottom-up) traversal and then notifies each composite widget with one or more managed children by means of its change_managed procedure.

Notifying a parent about its managed set involves geometry layout and possibly geometry negotiation. A parent deals with constraints on its size imposed from above (for example, when a user specifies the application window size) and suggestions made from below (for example, when a primitive child computes its preferred size). One difference between the two can cause geometry changes to ripple in both directions through the widget tree. The parent may force some of its children to change size and position and may issue geometry requests to its own parent in order to better accommodate all its children. You cannot predict where anything will go on the screen until this process finishes.

Appendix A

Resource File Format

A resource file contains text representing the default resource values for an application or set of applications.

The format of resource files is defined by *Xlib — C Language X Interface* and is reproduced here for convenience only.

The format of a resource specification is

```
ResourceLine     = Comment | IncludeFile | ResourceSpec | <empty line>
Comment          = “!” {<any character except null or newline>}
IncludeFile      = “#” WhiteSpace “include” WhiteSpace FileName WhiteSpace
FileName         = <valid filename for operating system>
ResourceSpec     = WhiteSpace ResourceName WhiteSpace “:” WhiteSpace Value
ResourceName     = [Binding] {Component Binding} ComponentName
Binding          = “.” | “*”
WhiteSpace       = {<space> | <horizontal tab>}
Component        = “?” | ComponentName
ComponentName    = NameChar {NameChar}
NameChar         = “a”-“z” | “A”-“Z” | “0”-“9” | “_” | “-”
Value            = {<any character except null or unescaped newline>}
```

Elements separated by vertical bar (|) are alternatives. Curly braces ({...}) indicate zero or more repetitions of the enclosed elements. Square brackets ([...]) indicate that the enclosed element is optional. Quotes (“...”) are used around literal characters.

If the last character on a line is a backslash (\), that line is assumed to continue on the next line.

To allow a Value to begin with whitespace, the two-character sequence “\space” (backslash followed by space) is recognized and replaced by a space character, and the two-character sequence “\tab” (backslash followed by horizontal tab) is recognized and replaced by a horizontal tab character.

To allow a Value to contain embedded newline characters, the two-character sequence “\n” is recognized and replaced by a newline character. To allow a Value to be broken across multiple lines in a text file, the two-character sequence “\newline” (backslash followed by newline) is recognized and removed from the value.

To allow a Value to contain arbitrary character codes, the four-character sequence “\nnn”, where each *n* is a digit character in the range of “0”−“7”, is recognized and replaced with a single byte that contains the octal value specified by the sequence. Finally, the two-character sequence “\\” is recognized and replaced with a single backslash.

Appendix B

Translation Table Syntax

Notation

Syntax is specified in EBNF notation with the following conventions:

```
[ a ]       Means either nothing or “a”
{ a }       Means zero or more occurrences of “a”
( a | b )   Means either “a” or “b”
\\n         Is the newline character
```

All terminals are enclosed in double quotation marks (“ ”). Informal descriptions are enclosed in angle brackets (< >).

Syntax

The syntax of a translation table is

```
translationTable  = [ directive ] { production }
directive         = ( “#replace” | “#override” | “#augment” ) “\\n”
production        = lhs “:” rhs “\\n”
lhs               = ( event | keyseq ) { “,” (event | keyseq) }
keyseq            = “"” keychar {keychar} “"”
keychar           = [ “^” | “$” | “\\” ] <ISO Latin 1 character>
event             = [modifier_list] “<”event_type“>” [ “(” count[“+”] “)” ] {detail}
modifier_list     = ( [“!”] [“:”] {modifier} ) | “None”
modifier          = [“~”] modifier_name
count             = (“1” | “2” | “3” | “4” | ...)
modifier_name     = “@” <keysym> | <see ModifierNames table below>
event_type        = <see Event Types table below>
detail            = <event specific details>
rhs               = { name “(” [params] “)” }
name              = namechar { namechar }
namechar          = { “a”-“z” | “A”-“Z” | “0”-“9” | “_” | “-” }
params            = string {“,” string}
string            = quoted_string | unquoted_string
quoted_string     = “"” {<Latin 1 character> | escape_char} [“\\\\” ] “"”
escape_char       = “\\"”
unquoted_string   = {<Latin 1 character except space, tab, “,”, “\\n”, “)”>}
```

The *params* field is parsed into a list of **String** values that will be passed to the named action procedure. A *quoted string* may contain an embedded quotation mark if the quotation mark is preceded by a single backslash (\). The three-character sequence “\\"” is interpreted as “single backslash followed by end-of-string”.

Modifier Names

The modifier field is used to specify standard X keyboard and button modifier mask bits. Modifiers are legal on event types **KeyPress**, **KeyRelease**, **ButtonPress**, **ButtonRelease**, **MotionNotify**, **EnterNotify**, **LeaveNotify**, and their abbreviations. An error is generated when a translation table that contains modifiers for any other events is parsed.

- If the modifier list has no entries and is not ‘‘None’’, it means ‘‘don’t care’’ on all modifiers.
- If an exclamation point (!) is specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and no other modifiers can be asserted.
- If any modifiers are specified and an exclamation point (!) is not specified, it means that the listed modifiers must be in the correct state and ‘‘don’t care’’ about any other modifiers.
- If a modifier is preceded by a tilde (˜), it means that that modifier must not be asserted.
- If ‘‘None’’ is specified, it means no modifiers can be asserted.
- If a colon (:) is specified at the beginning of the modifier list, it directs the Intrinsics to apply any standard modifiers in the event to map the event keycode into a KeySym. The default standard modifiers are Shift and Lock, with the interpretation as defined in *X Window System Protocol*, Section 5. The resulting KeySym must exactly match the specified KeySym, and the nonstandard modifiers in the event must match the modifier list. For example, ‘‘:<Key>a’’ is distinct from ‘‘:<Key>A’’, and ‘‘:Shift<Key>A’’ is distinct from ‘‘:<Key>A’’.
- If both an exclamation point (!) and a colon (:) are specified at the beginning of the modifier list, it means that the listed modifiers must be in the correct state and that no other modifiers except the standard modifiers can be asserted. Any standard modifiers in the event are applied as for colon (:) above.
- If a colon (:) is not specified, no standard modifiers are applied. Then, for example, ‘‘<Key>A’’ and ‘‘<Key>a’’ are equivalent.

In key sequences, a circumflex (ˆ) is an abbreviation for the Control modifier, a dollar sign ($) is an abbreviation for Meta, and a backslash (\) can be used to quote any character, in particular a double quote ("), a circumflex (ˆ), a dollar sign ($), and another backslash (\). Briefly:

No modifiers:	None <event> detail
Any modifiers:	<event> detail
Only these modifiers:	! mod1 mod2 <event> detail
These modifiers and any others:	mod1 mod2 <event> detail

The use of ‘‘None’’ for a modifier list is identical to the use of an exclamation point with no modifers.

Modifier	Abbreviation	Meaning
Ctrl	c	Control modifier bit
Shift	s	Shift modifier bit
Lock	l	Lock modifier bit
Meta	m	Meta key modifier
Hyper	h	Hyper key modifier
Super	su	Super key modifier
Alt	a	Alt key modifier
Mod1		Mod1 modifier bit
Mod2		Mod2 modifier bit
Mod3		Mod3 modifier bit
Mod4		Mod4 modifier bit
Mod5		Mod5 modifier bit
Button1		Button1 modifier bit
Button2		Button2 modifier bit
Button3		Button3 modifier bit

Modifier	Abbreviation	Meaning
Button4		Button4 modifier bit
Button5		Button5 modifier bit
None		No modifiers
Any		Any modifier combination

A key modifier is any modifier bit one of whose corresponding KeyCodes contains the corresponding left or right KeySym. For example, ‘‘m’’ or ‘‘Meta’’ means any modifier bit mapping to a KeyCode whose KeySym list contains XK_Meta_L or XK_Meta_R. Note that this interpretation is for each display, not global or even for each application context. The Control, Shift, and Lock modifier names refer explicitly to the corresponding modifier bits; there is no additional interpretation of KeySyms for these modifiers.

Because it is possible to associate arbitrary KeySyms with modifiers, the set of key modifiers is extensible. The ‘‘@’’ <keysym> syntax means any modifier bit whose corresponding KeyCode contains the specified KeySym name.

A modifier_list/KeySym combination in a translation matches a modifiers/KeyCode combination in an event in the following ways:

1. If a colon (:) is used, the Intrinsics call the display’s **XtKeyProc** with the KeyCode and modifiers. To match, (*modifiers* & ˜*modifiers_return*) must equal *modifier_list*, and *keysym_return* must equal the given KeySym.
2. If (:) is not used, the Intrinsics mask off all don’t-care bits from the modifiers. This value must be equal to *modifier_list*. Then, for each possible combination of don’t-care modifiers in the modifier list, the Intrinsics call the display’s **XtKeyProc** with the KeyCode and that combination ORed with the cared-about modifier bits from the event. *Keysym_return* must match the KeySym in the translation.

Event Types

The event-type field describes XEvent types. In addition to the standard Xlib symbolic event type names, the following event type synonyms are defined:

Type	Meaning
Key	**KeyPress**
KeyDown	**KeyPress**
KeyUp	**KeyRelease**
BtnDown	**ButtonPress**
BtnUp	**ButtonRelease**
Motion	**MotionNotify**
PtrMoved	**MotionNotify**
MouseMoved	**MotionNotify**
Enter	**EnterNotify**
EnterWindow	**EnterNotify**
Leave	**LeaveNotify**
LeaveWindow	**LeaveNotify**
FocusIn	**FocusIn**
FocusOut	**FocusOut**
Keymap	**KeymapNotify**
Expose	**Expose**
GrExp	**GraphicsExpose**

Type	Meaning
NoExp	**NoExpose**
Visible	**VisibilityNotify**
Create	**CreateNotify**
Destroy	**DestroyNotify**
Unmap	**UnmapNotify**
Map	**MapNotify**
MapReq	**MapRequest**
Reparent	**ReparentNotify**
Configure	**ConfigureNotify**
ConfigureReq	**ConfigureRequest**
Grav	**GravityNotify**
ResReq	**ResizeRequest**
Circ	**CirculateNotify**
CircReq	**CirculateRequest**
Prop	**PropertyNotify**
SelClr	**SelectionClear**
SelReq	**SelectionRequest**
Select	**SelectionNotify**
Clrmap	**ColormapNotify**
Message	**ClientMessage**
Mapping	**MappingNotify**

The supported abbreviations are:

Abbreviation	Event Type	Including
Ctrl	**KeyPress**	with Control modifier
Meta	**KeyPress**	with Meta modifier
Shift	**KeyPress**	with Shift modifier
Btn1Down	**ButtonPress**	with Button1 detail
Btn1Up	**ButtonRelease**	with Button1 detail
Btn2Down	**ButtonPress**	with Button2 detail
Btn2Up	**ButtonRelease**	with Button2 detail
Btn3Down	**ButtonPress**	with Button3 detail
Btn3Up	**ButtonRelease**	with Button3 detail
Btn4Down	**ButtonPress**	with Button4 detail
Btn4Up	**ButtonRelease**	with Button4 detail
Btn5Down	**ButtonPress**	with Button5 detail
Btn5Up	**ButtonRelease**	with Button5 detail
BtnMotion	**MotionNotify**	with any button modifier
Btn1Motion	**MotionNotify**	with Button1 modifier
Btn2Motion	**MotionNotify**	with Button2 modifier
Btn3Motion	**MotionNotify**	with Button3 modifier
Btn4Motion	**MotionNotify**	with Button4 modifier
Btn5Motion	**MotionNotify**	with Button5 modifier

The detail field is event-specific and normally corresponds to the detail field of the corresponding event as described by *X Window System Protocol*, Section 11. The detail field is supported for the following event types:

Event	Event Field
KeyPress	KeySym from event *detail* (keycode)
KeyRelease	KeySym from event *detail* (keycode)
ButtonPress	button from event *detail*
ButtonRelease	button from event *detail*
MotionNotify	event *detail*
EnterNotify	event *mode*
LeaveNotify	event *mode*
FocusIn	event *mode*
FocusOut	event *mode*
PropertyNotify	*atom*
SelectionClear	*selection*
SelectionRequest	*selection*
SelectionNotify	*selection*
ClientMessage	*type*
MappingNotify	*request*

If the event type is **KeyPress** or **KeyRelease**, the detail field specifies a KeySym name in standard format which is matched against the event as described above, for example, <Key>A.

For the **PropertyNotify**, **SelectionClear**, **SelectionRequest**, **SelectionNotify**, and **ClientMessage** events the detail field is specified as an atom name; for example, <Message>WM_PROTOCOLS. For the **MotionNotify**, **EnterNotify**, **LeaveNotify**, **FocusIn**, **FocusOut**, and **MappingNotify** events, either the symbolic constants as defined by *X Window System Protocol*, Section 11, or the numeric values may be specified.

If no detail field is specified, then any value in the event detail is accepted as a match.

A KeySym can be specified as any of the standard KeySym names, a hexadecimal number prefixed with “0x” or “0X”, an octal number prefixed with “0”, or a decimal number. A KeySym expressed as a single digit is interpreted as the corresponding Latin 1 KeySym, for example, “0” is the KeySym XK_0. Other single character KeySyms are treated as literal constants from Latin 1, for example, “!” is treated as 0x21. Standard KeySym names are as defined in **<X11/keysymdef.h>** with the “XK_” prefix removed.

Canonical Representation

Every translation table has a unique, canonical text representation. This representation is passed to a widget’s **display_accelerator** procedure to describe the accelerators installed on that widget. The canonical representation of a translation table is (see also “Syntax”)

```
translationTable   = { production }
production         = lhs “:” rhs “\\n”
lhs                = event { “,” event  }
event              = [modifier_list] “<”event_type“>” [ “(” count[“+”] “)” ] {detail}
modifier_list      = [“!”] [“:”] {modifier}
modifier           = [“~”] modifier_name
count              = (“1” | “2” | “3” | “4” | ...)
modifier_name      = “@” <keysym> | <see canonical modifier names below>
event_type         = <see canonical event types below>
detail             = <event-specific details>
rhs                = { name “(” [params] “)” }
name               = namechar { namechar }
namechar           = { “a”-“z” | “A”-“Z” | “0”-“9” | “_” | “-” }
```

params	= string {“,” string}
string	= quoted_string
quoted_string	= “"” {<Latin 1 character> | escape_char} [“\\\\”] “"”
escape_char	= “\\"”

The canonical modifier names are

Ctrl	Mod1	Button1
Shift	Mod2	Button2
Lock	Mod3	Button3
	Mod4	Button4
	Mod5	Button5

The canonical event types are

KeyPress	**KeyRelease**
ButtonPress	**ButtonRelease**
MotionNotify	**EnterNotify**
LeaveNotify	**FocusIn**
FocusOut	**KeymapNotify**
Expose	**GraphicsExpose,**
NoExpose	**VisibilityNotify**
CreateNotify	**DestroyNotify**
UnmapNotify	**MapNotify**
MapRequest	**ReparentNotify**
ConfigureNotify	**ConfigureRequest**
GravityNotify	**ResizeRequest**
CirculateNotify	**CirculateRequest**
PropertyNotify	**SelectionClear**
SelectionRequest	**SelectionNotify**
ColormapNotify	**ClientMessage**

Examples

- Always put more specific events in the table before more general ones:

```
Shift <Btn1Down> : twas()\n\
<Btn1Down> : brillig()
```

- For double-click on Button1 Up with Shift, use this specification:

```
Shift<Btn1Up>(2) : and()
```

  This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down>,Shift<Btn1Up> : and()
```

- For double-click on Button1 Down with Shift, use this specification:

```
Shift<Btn1Down>(2) : the()
```

  This is equivalent to the following line with appropriate timers set between events:

```
Shift<Btn1Down>,Shift<Btn1Up>,Shift<Btn1Down> : the()
```

- Mouse motion is always discarded when it occurs between events in a table where no motion event is specified:

  <Btn1Down>,<Btn1Up> : slithy()

  This is taken, even if the pointer moves a bit between the down and up events. Similarly, any motion event specified in a translation matches any number of motion events. If the motion event causes an action procedure to be invoked, the procedure is invoked after each motion event.
- If an event sequence consists of a sequence of events that is also a noninitial subsequence of another translation, it is not taken if it occurs in the context of the longer sequence. This occurs mostly in sequences like the following:

  <Btn1Down>,<Btn1Up> : toves()\n\
  <Btn1Up> : did()

  The second translation is taken only if the button release is not preceded by a button press or if there are intervening events between the press and the release. Be particularly aware of this when using the repeat notation, above, with buttons and keys, because their expansion includes additional events; and when specifying motion events, because they are implicitly included between any two other events. In particular, pointer motion and double-click translations cannot coexist in the same translation table.
- For single click on Button1 Up with Shift and Meta, use this specification:

  Shift Meta <Btn1Down>, Shift Meta<Btn1Up>: gyre()

- For multiple clicks greater or equal to a minimum number, a plus sign (+) may be appended to the final (rightmost) count in an event sequence. The actions will be invoked on the *count*-th click and each subsequent one arriving within the multi-click time interval. For example:

  Shift <Btn1Up>(2+) : and()

- To indicate **EnterNotify** with any modifiers, use this specification:

  <Enter> : gimble()

- To indicate **EnterNotify** with no modifiers, use this specification:

  None <Enter> : in()

- To indicate **EnterNotify** with Button1 Down and Button2 Up and "don't care" about the other modifiers, use this specification:

  Button1 ~Button2 <Enter> : the()

- To indicate **EnterNotify** with Button1 down and Button2 down exclusively, use this specification:

  ! Button1 Button2 <Enter> : wabe()

  You do not need to use a tilde (~) with an exclamation point (!).

-- --

Table of Contents